ICE: A Passive, High-Speed, State-Continuity Scheme

Raoul Strackx
iMinds-Distrinet, KU Leuven
Celestijnenlaan 200A
3001 Heverlee

Bart Jacobs
iMinds-Distrinet, KU Leuven
Celestijnenlaan 200A
3001 Heverlee
first.last@cs.kuleuven.be

Frank Piessens
iMinds-Distrinet, KU Leuven
Celestijnenlaan 200A
3001 Heverlee

ABSTRACT

The amount of trust that can be placed in commodity computing platforms is limited by the likelihood of vulnerabilities in their huge software stacks. Protected-module architectures, such as Intel SGX, provide an interesting alternative by isolating the execution of software modules. To minimize the amount of code that provides support for the protected-module architecture, persistent storage of (confidentiality and integrity protected) states of modules can be delegated to the untrusted operating system. But precautions should be taken to ensure *state continuity*: an attacker should not be able to cause a module to use stale states (a so-called *rollback attack*), and while the system is not under attack, a module should always be able to make progress, even when the system could crash or lose power at unexpected, random points in time (i.e., the system should be *crash resilient*).

Providing state-continuity support is non-trivial as many algorithms are vulnerable to attack, require on-chip non-volatile memory, wear-out existing off-chip secure non-volatile memory and/or are too slow for many applications.

We present ICE, a system and algorithm providing state-continuity guarantees to protected modules. ICE's novelty lies in the facts that (1) it does not rely on secure non-volatile storage for every state update (e.g., the slow TPM chip). (2) ICE is a passive security measure. An attacker interrupting the main power supply or any other source of power, cannot break state-continuity. (3) Benchmarks show that ICE already enables state-continuous updates almost 5x faster than writing to TPM NVRAM. With dedicated hardware, performance can be increased 2 orders of magnitude.

ICE's security properties are guaranteed by means of a machine-checked proof and a prototype implementation is evaluated on commodity hardware.

1. INTRODUCTION

Protection of sensitive data in commodity computing platforms is extremely challenging. Modern operating systems provide process isolation primitives, but the kernel is too large to be implemented free from vulnerabilities. Moreover, commodity systems are prone to physical attacks, even by ill-equipped and resource-constrained home users. These vulnerabilities limit the amount of trust that can be placed in commodity systems. In servers these limitations are remedied by programmable hardware security modules (HSMs). On client devices, highly-sensitive applications such as online banking or e-government often resort to smart cards. Unfortunately, these solutions are expensive, cumbersome and the security guarantees that they can provide to the overall applications are limited.

Two recent advances in computer security indicate that this situation may change in the near future. First, protected-module architectures (PMAs) have been developed that provide strong isolation directly to modules running at application level [3, 4, 16, 17, 19, 28, 29, 36]. The OS is still relied upon to provide services such as disk and network access, but they are *not* trusted. Protected modules' memory regions cannot be accessed from unprotected memory; modules are in complete control over their own content and can only be accessed through the interface they expose. Last year Intel disclosed their work on Software Guard eXtension (SGX) [2, 9, 18], their own hardware-implemented PMA for commodity processors. SGX goes even further than other state-of-the-art PMAs and also provides protection against hardware attacks; modules (called enclaves in SGX[1]) are only stored in plaintext within the CPU package. When they are evicted to main memory they are confidentiality, integrity and version protected.

Second, Agten et al. [1] and Patrignani et al. [21, 22] proposed fully-abstract compilation techniques to protected-module architectures. While the strong isolation guarantees offered by these architectures is vital, they are difficult to implement without compiler support. Care must be taken not to introduce software vulnerabilities during compilation. Fully-abstract compilation ensures just this; machine-code-level attacks exists *iff* also a corresponding attack at source-code level exists. This enables easy reasoning and verification of the security guarantees these modules provide.

Unfortunately an important attack vector has been largely overlooked. Protected-module architectures, including SGX, only provide strong isolation guarantees *while the system executes continuously*. Without support for state continuity, protected modules need to remain stateless, significantly

ACSAC '14, December 08–12, 2014, New Orleans, LA, USA.

ACM 978-1-4503-3005-3/14/12 ...$15.00.
http://dx.doi.org/10.1145/2664243.2664259 .

[1]We will use the term "protected module" when referring to isolated memory areas in *any* protected-module architecture and use "enclave" when referring to SGX specifically.

hampering their applicability. Consider as a running example a password-checking module. To defend against dictionary attacks, the user will be locked out indefinitely after three failed attempts. The module confidentiality and integrity protects its state before handing it to the untrusted operating system for storage. But when the module needs to recover its state after a reboot, it cannot distinguish between a fresh and a stale state and the guess-limited security measure cannot be guaranteed.

In practice most applications and protocols rely on state-continuity guarantees; firewall settings most not be revertible, attackers must not be able to tamper with log files, revoked user credentials must not be rolled back, cryptographic nonces must never be re-used, etc. Support for state continuity may also provide stronger security guarantees. Chun et al. for example proposed append-only memory [5] to harden existing distributed algorithms and applications such as NFS. Acting as a trusted log, this memory protects against equivocation; the ability of a network node to make contradicting statements to different entities.

While at first glance having similarities with replay attacks, the state itself is replayed in a rollback attack. Providing support for state continuity is therefore much harder, especially when practical limitations are considered. Parno et al. [20] show that many seemingly obvious algorithms are flawed. Others are prone to simple hardware attacks. Attaching an uninterruptible power source (UPS), for example, may simply be disconnected. Or an in-kernel attacker may prevent the execution of the interrupt handlers it relies upon. Adding non-volatile memory on-chip could simplify a solution, but requires modification of manufacturing processes leading to increased manufacturing costs. Alternatively, using non-volatile memory off-chip (e.g., isolating disk space) may be susceptible to a clone attack where a hardware-level attacker may easily overwrite the state with a previously recorded stale state. Using TPM NVRAM or TPM monotonic counters instead, would foil such attacks, but would significantly impact performance and usability. Most implementations only provide 1,280 bytes of NVRAM that supports only 100,000 write cycles over the chip's lifetime [20]. Accessing NVRAM every second, would wear it out in less then 28 hours. Monotonic counters, on the other hand, only need to be incrementable every 5 seconds [35].

Hardware upgrades to the TPM chip could reduce some of these architectural constraints, at an economic cost. However, any solution placing the TPM on the performance-critical path, would require additional upgrades over time to bridge the ever growing TPM/CPU performance gap. We present ICE, an alternative solution that only requires TPM accesses at boot time and is thus *not* affected by TPM speed.

ICE avoids architectural challenges (1) by proposing a simple implementation technique where on-chip dedicated registers are backed off-chip by a capacitor and persistent memory. Upon a sudden loss of power, the contents of the dedicated registers is written to persistent memory. (2) ICE is a *passive* protection scheme; in the event of a crash or power loss, security is guaranteed instantly. A hardware attacker may disconnect the capacitor, but state continuity remains guaranteed. (3) At the moment freshness information is backed to persistent storage, it is considered public data. Overwriting it with stale freshness information will be detected upon recovery.

In summary, we make the following contributions:

- We present ICE, the first algorithm providing state-continuity guarantees with a minimal TCB that does *not* rely on the speed of secure, non-volatile memory (e.g., the (slow) TPM chip) nor does it rely on an uninterruptible power source.
- We formally verify and machine check the security properties of ICE using the Coq proof assistant.
- Because SGX-enabled machines or emulators are not yet available, we validate our claims based on a prototype implementation on top of Fides [28], an existing hypervisor-based protected module architecture similar to SGX. Benchmarks show that states can already be stored almost 5x faster on commodity hardware than writing to TPM NVRAM. Dedicated hardware support would increase performance substantially.

The remainder of this paper is structured as follows. First we detail our attack model and the security properties that we need to guarantee. Next in Sections 3 and 4, we present our algorithm and discuss two possible implementations. Finally, we evaluate the security and performance of ICE.

2. PROBLEM DEFINITION

2.1 Attacker Model

ICE can defend against an attacker with three powerful capabilities. First, we assume that an attacker is able to compromise the entire software stack, with the exception of ICE-implementing modules. This enables versatile attacks ranging from modifying the contents of the hard drive to preventing enclaves from ever resuming execution.

Second, we assume that an attacker has control over the system's power supply or is able to launch attacks leading to a similar result. Power-interruption attacks differ from kernel-level crashes as they also affect software modules executing in complete isolation from the rest of the system: modules may stop executing before they can commit their new state. SGX enclaves are especially vulnerable to such attacks. In order to prevent denial-of-service attacks by malicious enclaves that never return control to the kernel, SGX supports interruption of enclaves [11]. When the interrupt is handled in the untrusted kernel, an in-kernel attacker can easily prevent the enclave from ever resuming execution.

Third, we consider hardware attacks. We implement ICE as a library that modules can be statically linked with and take advantage of the security guarantees provided by the protected-module architecture. In case of SGX this implies that an attacker may place probes on memory buses or perform cold boot attacks [8]. Defending against physical attacks against the CPU package itself or the TPM chip [26,32,38] are orthogonal problems and not considered.

2.2 Security Properties

State continuity can be factored into two properties: safety and liveness. To ensure safety, ICE must be resilient against a *rollback attack* where an attacker provides the module with a valid, but stale state. A rollback attack is related to a replay attack but it is much harder to defend against. Where in a replay attack identical input is provided, the state of the module itself is replayed in a rollback attack.

The second property, liveness, states that benign events should never force the system into a state from which it can-

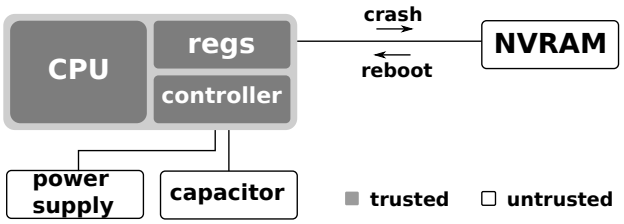

Figure 1: Architecture of guarded memory. When power suddenly fails on-chip dedicated registers are backed up to off-chip, shadow memory (NVRAM).

not progress. In practice this means that the system should be allowed to crash at any time during the operation of the algorithm, including when it is recovering from a previous crash. Note that this is not the same as protection against denial-of-service. Protection against denial-of-service is not in scope; in-kernel attackers can easily prevent the system from progressing (e.g., by breaking the kernel). Liveness only ensures progress is not hampered by *random* crashes, which may also occur when the system is not under attack.

3. STATE-CONTINUITY AS A LIBRARY

Before introducing a running example and describing ICE in full detail, we first introduce the system hardware we rely on and discuss how freshness information is recorded.

3.1 Architecture

Assuming ICE is implemented on top of Intel SGX, we only place trust in the CPU package and TPM chip. Attacks against any other component cannot compromise security.

Enclaves.

Intel SGX provides enclaves with total control over their own code and data by enforcing a specific access control mechanism; *only* when executing within the boundaries of an enclave can its content be accessed. Access attempts from code running at *any* privilege level outside the enclave (including from other enclaves), will be blocked. Enclaves can only be accessed through an explicitly exposed interface.

TPM.

We store long term secrets and freshness information in TPM NVRAM. These secrets should only be accessible from the SGX enclave that provided them.

Guarded Memory.

To enable fast state updates, we propose the addition of a small amount of *guarded memory*; dedicated registers on-chip that are backed off-chip by shadow, non-volatile memory (NVRAM) and a capacitor (see Fig. 1). When a controller detects that the main power supply is disconnected from the CPU package, it copies the registers' content to non-volatile memory. When power is re-applied, the registers are restored. Note that *only* on-chip components need to be trusted. Attacks against shadow memory, main power supply or the capacitor cannot break state continuity.

Persistent Storage.

ICE uses operating system services to access persistent storage. These services are not trusted: an attacker may copy, replace and destroy files. To differentiate between the actual state of a module and states stored on disk, we call the latter *(ICE) cubes* whenever ambiguity might arise.

3.2 Guards: Storing Freshness Info

Just as message authentication codes (MACs) can be used to guarantee message integrity, we will use *guards* to prove that a cube is fresh. Guards are 2-tuples:

$$\text{guard}_i(n) = (\underbrace{\text{Hash}^i(n)}_{\text{guard value}}, \underbrace{i}_{\text{guard index}})$$

where the first element, the *guard value* represents the hash value after hashing the base value i times, the *guard index*.

A guard is incremented by hashing the guard value and incrementing the index:

$$\text{guard}_i(n) = (\text{Hash}^i(n), i)$$
$$\text{guard}_{i+1}(n) = (\text{Hash}^{i+1}(n), i+1)$$

Based on the construction of guards, they possess two important properties: (1) two guards can be compared based on the guard index:

$$(n,i) \leq (m,j) \Leftrightarrow \begin{cases} n = m & \text{if } i = j \\ (\text{Hash}(n), i+1) \leq (m,j) & \text{if } i < j \end{cases}$$

and, (2) an attacker is unable to calculate any preceding guard as this would imply inverting the hash function.

3.3 ChkPassword: A Running Toy Example

Guaranteeing state-continuity is non-trivial and can only be accomplished by a module provider taking the required safety precautions. We only provide a library offering state-continuous storage. To demonstrate the subtle vulnerabilities that need to be resolved, consider as a running example `ChkPassword`, a password-checking module displayed in listing 1. It exposes an interface of two functions: `set_passwd` that modifies the user's password and `check_passwd`[2] that handles login attempts. To prevent dictionary attacks, `ChkPassword` will lock out a user indefinitely after 3 incorrect attempts. We assume that when the module is created, the `INIT` function is called before any service call is handled. When `ChkPassword` executes on the platform for the first time, a default password is selected (line 7), otherwise its previous state is restored (line 10).

To ensure state continuity, `ChkPassword` needs to fulfill three requirements. First, it must protect against subtle timing attacks. When an attacker is able to infer that the provided password is incorrect based on timing differences between a correct and incorrect password[3], she may be able to crash the system before the login attempt could be recorded. Ensuring that each execution path takes exactly the same amount of CPU cycles is hard. Similar to Parno et al. [20], we take a much simpler approach and store the state with the newly provided input *before* it is used in any computation. Hence, `ChkPassword` stores its current state

[2]Calling `ChkPassword` from unprotected memory would enable an attacker to intercept the provided password before it reaches the module. Users of `ChkPassword` should establish a secure channel from another module before exchanging sensitive data [2, 28]. Such considerations are out of scope.

[3]A similar attack exists when a (unique) callback to unprotected memory is made before an undesirable state is stored.

```
1  static int attempts_left;
   static char *password;
3
   void INIT( void ) {
5    State *state;
     if (retrieve( &state ) == UNINITIALIZED){
7        password = "default";
         attempts_left = 3;
9    } else
         restore_and_restart( state );
11 }

13 int ENTRY_POINT check_passwd(char *guess) {
     State *state = new State();
15
     //store (input, state) tuple
17   collect_state( state );
     collect_entry( state, "CHECK_PASSWD" );
19   collect_input( state, guess );
     store( state );
21
     //check passwd
23   if ( attempts_left > 0 &&
          strcmp( password, guess ) == 0 ) {
25     attempts_left = 3;
       return OK;
27   } else {
       attempts_left = max(attempts_left -1,0);
29     return INCORRECT;
     }
31 }

33 int ENTRY_POINT set_passwd(char *oldpwd,
       char *newpwd) { ... }
```

Listing 1: ChkPassword: A running example

(the number of attempts left and the correct password) together with the provided guess (line 17-20) before checking the provided password. An unexpected crash while the password is being verified (i.e., after line 20), will then result in the current state being restored and execution is restarted; another attempt is made to check the *same* provided password. We assume `restore_and_restart` restores the current state and restarts execution of the last called entry point (line 10). Alternatively, if the system crashed before the input could be recorded and thus was never used in any meaningful computation (i.e., before line 20), the provided guess can simply be discarded.

Second, in order to guarantee that re-execution of the same input on the same state always leads to an identical result, modules must be deterministic. This implies that modules must consider all sources of non-determinism (e.g., the result of a random number generator) as input and thus store such data before using it in any computation.

Third, an attacker must not be able to infer any value from the size of the stored states on disk; modules must ensure that all cubes are equal in size.

3.4 ICE Libraries

We will provide state-continuous storage in two steps. In Section 3.4.1 we introduce `libice0`, a library providing support at the cost of scarce platform resources for every instance. Then in Section 3.4.2, we present `libicen` that alleviates resource pressure by storing freshness information

```
1  void store( State *state ) {
     switch (ice.mode) {
3      case Clear:
         return _init_state( state );
5      case Activated:
         return _update_state( state );
7  } }

9  int retrieve( State **state ) {
     switch (tpm.mode) {
11     case Clear:
         return UNINITIALIZED;
13     case Activated:
         *state = _recovery_step();
15       return RECOVERED;
   } }
```

Listing 2: `libice0` relies on `tpm.mode` and `ice.mode` to distinguish between storing an initial state, updating a stored state and recovery

in a single, state-continuous module `ice0`. As all `libicen` library instances connect to the same, unique `ice0` instance, a virtually unlimited number of modules is supported.

Both `libice0` and `libicen` provide the same interface: `store(State *)` and `retrieve(State **)`. To avoid repeated TPM or `ice0` accesses, `libice0` and `libicen` keep a cached copy. In order to distinguish between these copies and explicitly state where they are stored, we will reference them similarly to fields of a struct. For example, the encryption and MAC keys stored in the TPM chip will be referenced as `tpm.keys`. The variables used by the ICE algorithm are referenced as `ice.keys` and so on. Besides storing keys and the guard we also keep track of the state of the algorithm using a `mode` variable. Stored inside the TPM chip (`tpm.mode`), this variable indicates whether ICE was once initiated correctly. In `libice0` (`ice.mode`) this variable is used to indicate whether ICE was initiated or recovered since reboot. We assume that when a module is resurrected after a crash, `ice.mode` is initialized with value `Clear`. As a shorthand, we also assume that setting this variable takes exclusive access of guarded memory. Listing 2 uses these variables to differentiate between an initial state being stored and a state being updated. Similarly, `tpm.mode` is used to determine whether a state was ever stored.

3.4.1 `libice0`: State-Cont. Storage for One Module

In order to provide state continuity, we must guarantee that an attacker is not able to fabricate recorded states (called cubes) and that no stale cubes can be provided as being fresh. The former is trivially guaranteed by including a message authentication code in each cube. Guaranteeing freshness is more challenging, but as modules maintain their state between invocations, we only need to consider power off and reboot events. Let's call events during such power cycles an *execution stream*. An execution stream starts by either storing an initial state of a module or when the state of a module is recovered after a crash. It ends when the system crashes or when it is shut down properly.

To keep track of the fresh cube, we will generate a (base) guard when the execution stream starts and store it securely in TPM NVRAM. For every state the module requests storage of in the current execution stream, we will increment the

```
void _init_state( State *state ) {
  ice.guard = gen_guard();
  ice.keys = gen_keys();
  hdd.write( new Cube( ice.guard, ice.keys,
       state ));
  ice.mode = Activated;
  gmem.guard = ice.guard;
  tpm.guard = ice.guard;
  tpm.keys = ice.keys;
  tpm.mode = Activated;
}
```

Listing 3: `libice0`: Storing the initial state

```
void _update_state( State *state ) {
  ice.guard = ++ice.guard;
  hdd.write( new Cube( ice.guard, ice.keys,
       state ));
  gmem.guard = ice.guard;
}
```

Listing 4: `libice0`: Updating a state

guard and include it in the generated cube. Using guarded memory we will ensure that *only* the guard included in the last (and thus fresh) cube is leaked at the moment the system crashes. As no preceding guards were leaked (and cannot be calculated), it serves as a pointer to the fresh cube. Upon recovery, knowledge of the guard[4] that is stored in the provided cube, proves that the cube is fresh.

Creation of an Initial State.

When storage of the initial state of the module is requested, a new base guard and keys are generated (see listing 3). Next, a new cube is constructed and written to disk. Exclusive access of guarded memory is taken by setting the `ice.mode` variable to `Activated` and the fresh guard is written to guarded memory. In case exclusive access cannot be assigned (i.e., another module already received it), the module simply stops its execution. For clarity, such error handling is not displayed. Finally the keys and guard are stored in the TPM's NVRAM and `tpm.mode` is set to `Activated`, committing the start of a new execution stream.

Updating a State.

When storage of a new input-state pair is requested in the same execution stream, the previously used guard and keys are still stored in `libice0`'s memory and no TPM accesses are required. To safely store the input-state pair, a new cube is created with the subsequent guard and stored on disk (listing 4). Finally the fresh guard is written to guarded memory, committing the step.

Recovering from a Crash.

Recovering from a crash is more challenging and is performed in two steps (see listing 5, error handling is omitted for clarity). First, the last stored cube is read from disk. By verifying three properties its freshness is ensured:

[4]We must also check that this guard was created during the last execution stream as a matching guard/cube is found at the end of every execution stream.

```
1  State *_recovery_step() {
     Cube cube = hdd.read();
3    if ( is_fresh( &cube ) ) {
       State *state = extract(cube, tpm.keys);
5      ice.guard = gen_guard();
       ice.keys = tpm.keys;
7      hdd.write( new Cube( ice.guard,
         ice.keys, state ) );
9      ice.mode = Activated;
       gmem.guard = ice.guard;
11     tpm.guard = ice.guard;
       return state;
13   }
     else abort();
15 }

17 bool is_fresh( Cube *cube ) {
     return ( check_mac( cube, tpm.keys ) &&
19     tpm.guard ≤ cube->guard &&
       gmem.guard.value == cube->guard.value);
21 }

23 bool operator≤( Guard g1, Guard g2 ) {
     while ( g1.index < g2.index ) {
25     g1.value = Hash( g1.value );
       ++g1.index;
27   }
     return g1.value == g2.value;
29 }
```

Listing 5: `libice0`: Recovering from a crash.

- *Validity*: Cubes must not have been forged. This is ensured by the MAC stored in each cube and the accompanying key stored securely in the TPM (line 17).
- *Correct execution stream*: The cube received from the untrusted OS must have been created during the last execution stream. Starting each execution stream, a new base guard is generated and stored safely in TPM NVRAM. All guards used during this execution stream are successors of this base guard. Hence, the cube was created during the last execution stream iff (line 18):

$$\text{tpm.guard} \leq \text{cube.guard}$$

- *Public guard*: `libice0` ensures that guarded memory always contains the same guard as the last (fresh) cube stored on disk[5], and that no preceding guards leak or can be calculated. Hence, if the guard stored in guarded memory matches the guard included in the cube at hand and the two previous properties hold as well, it is guaranteed that the cube is fresh (line 19).

In the second step the fresh state is re-stored as part of a new execution stream: `libice0`'s variables are restored from TPM NVRAM, a new base guard is generated, the fresh state packaged in a new cube and the base guard is written to guarded and TPM NVRAM memory. To ensure that after an unexpected crash during the execution of this step, recovery can be restarted, `libice0` must (1) backup the previous fresh guard before overwriting it in guarded

[5]There is one exception as writing cubes to disk and updating guarded memory cannot be executed atomically. This exception is resolved later in this section.

memory. As this value is public, any persistent storage can be used (for clarity not displayed in listing 5). (2) The new base guard is written to TPM NVRAM as the last step.

Let's reconsider `ChkPassword` and discuss how crashes are resolved. Depending on the timing of a crash, we can differentiate between three main situations. One, `ChkPassword` was just created and the user called `set_passwd` to change the default password. This led to the execution of `_init_state` but the system crashes before `tpm.mode` could be set (see listing 3, line 9). When `ChkPassword` is re-created, it requests its previous state (listing 1, line 6). As `tpm.mode` still read `Clear` (listing 2, line 11), the module will restart from its default settings. As no input was ever used, state-continuity is guaranteed trivially.

Two, the system didn't crash when the user modified the module's default password and now calls `check_passwd` providing `"attempt1"` as password. After `libice0` stores a new cube $C_{\texttt{attempt1}}$ on disk and updates guarded memory, the system crashes while the password is being verified (listing 1, line 23). The module is re-created and execution flow eventually executes `_recovery_step` (listing 5) As only a single cube is available containing the leaked guard from guarded memory (or a successor thereof), only cube $C_{\texttt{attempt1}}$ is considered fresh. After returning the stored input-state tuple in $C_{\texttt{attempt1}}$, `ChkPassword` will restore the `attempts_left` and `password` variables and execution is restarted with input `"attempt1"` (listing 1, line 10).

Three, assume that the previous password was incorrect and the user enters `"attempt2"` for her second attempt. After storing the new cube $C_{\texttt{attempt2}}$ on disk, the system crashes *before* the new (incremented) guard could be written to guarded memory (listing 4, line 4). This is an interesting point of failure as both cubes $C_{\texttt{attempt1}}$ as $C_{\texttt{attempt2}}$ can be considered fresh[6]. However, recovery based on either will preserve state continuity. This is obvious for cube $C_{\texttt{attempt2}}$ as this is the latest cube written to disk. Recovery from $C_{\texttt{attempt1}}$, however will purge any record of the login attempt made using `"attempt2"` as password. This is also safe as it was never used in any valuable computation (instructions after listing 1 line 24 were not executed yet). Hence, an attacker is not able to deduce any valuable information.

3.4.2 `libicen`: State-Cont. Storage for n Modules

By depending on scarce resources such as TPM NVRAM and guarded memory, `libice0` can in practice only provide state-continuous storage to a limited number of modules. `libicen` will alleviate this strain by using a single, unique `ice0` module to store freshness information on behalf of other modules. To safely exchange sensitive information between `libicen` and the `ice0` module, inter-module communication must guarantee endpoint authentication and confidentiality, integrity and freshness of messages. We will state this explicitly by passing a module identifier to `ice0` calls.

Creation of an initial state.

Similarly to `libice0`, an initial state of the module is stored by generating a new guard and cryptographic keys and writing a new cube to disk (see listing 6). Finally the `ice0` module is requested to store the keys and guard.

[6]The recovery step as displayed in listing 5, line 19, only accepts cube $C_{\texttt{attempt1}}$ as fresh. However, an attacker incrementing the guard stored in guarded memory, will trick `libice0` to accept cube $C_{\texttt{attempt2}}$ as being fresh as well.

```
1 void _init_step( State *state ) {
  mod.guard = gen_guard();
3 mod.keys = gen_keys();
  hdd.write( new Cube( mod.guard, mod.keys,
      state ));
5 mod.mode = Activated;
  ice0.store( mod.id, mod.keys, mod.guard);
7 }
```

Listing 6: `libicen`: Initialization of a new module

```
1 void _update_state( State *state ) {
  ++mod.guard;
3 hdd.write( new Cube( mod.guard, mod.keys,
      state ));
  ice0.store( mod.id, mod.guard );
5 }
```

Listing 7: `libicen`: Updating a state

Updating a state.

To update a state, `libicen` writes a new cube to disk, before the updated fresh guard is stored in `ice0` (see listing 7).

Recovering from a crash.

To recover from a crash, the (presumably) fresh cube is read from disk (see listing 8). Next, the keys and guard are requested from the `ice0` module. As the fresh guard is always stored safely in `ice0`, a cube with a correct MAC and that contains the fresh guard, must be fresh. Once the cube's freshness has been validated, `libicen` needs to generate a new guard, create and write a new cube to disk and store the new guard in `ice0` before a new step is taken. Storing the fresh cube with a newly generated guard is vital, even though the fresh guard never leaked. For details we refer the reader to the extended version of this paper [27].

4. IMPLEMENTATION

Given that SGX-enabled systems are not available yet, we implemented[7] ICE on top of Fides [28], a hypervisor-based protected-module architecture using CMOS memory as guarded memory. This setup enables microbenchmarks and detailed analysis of the costs of accessing the TPM chip, writing cubes to disk, performing cryptographic calculations and accessing guarded memory on *real-life* systems.

While Fides provides similar isolation mechanisms as Intel SGX, it cannot guarantee that protected modules, or its own implementation for that matter, do not leave the CPU package in plaintext. Hence, this setup cannot defend against a hardware attacker that is able to directly modify contents of main memory. Similarly, we assume that an attacker cannot modify contents of CMOS memory directly.

While *any* non-volatile memory can serve as an alternative, CMOS memory is an interesting candidate for guarded memory. As it already stores wall-clock time, it is updated every second and it must support a large number of write operations over its entire lifespan. Second, as it does not require a special communication protocol, it can be accessed easily and without much overhead. Being only accessible

[7]Our research prototype is available at `https://distrinet.cs.kuleuven.be/software/sce/`

```
State *_recovery_step() {
  Cube cube = hdd.read();
  ice0.retrieve( mod.id, &mod.keys, &mod.
      guard)
  if ( is_fresh( &cube ) ) {
    State *state = extract(cube.state, tpm.
        keys)
    mod.guard = gen_guard();
    hdd.write( new Cube( mod.guard, mod.
        keys, cube.state ) );
    mod.state = Activated;
    ice0.store(mod.id, mod.keys, mod.guard)
    return state;
  }
  else abort();
}

bool is_fresh( Cube *cube ) {
  return check_mac( cube, mod.keys ) &&
    mod.guard.value == cube.guard.value;
}
```

Listing 8: `libicen`: Recovering from a crash

	`libice0`		`libicen`	
	asm	C	asm	C
ICE	0	372	0	341
SHA-512	0	371	0	371
AES-NI	1,566	176	1,566	176
Total	1,566	919	1,566	888

Table 1: Breakdown of `libice0` and `libicen`.

through direct I/O, it can also be isolated easily by hardware virtualization support; only 21 lines of code (LOC) had to be added to the hypervisor. Another 61 LOCs were required to implement system calls to access CMOS memory from the `ice0` module. This totals the size of the hypervisor to 9,492 LOCs. While Fides at this moment does not support TPM chip accesses, we estimate, based on the Flicker [17] source code[8], that this straightforward effort would require an addition of less than 2,000 LOCs.

To implement `libice0` and `libicen`, we used the polarssl[9] library to calculate SHA-512 hash values and the Intel AES-NI reference implementation to take advantage of AES hardware support. This totals to 2,485 LOCs and 2,454 LOCs for `libice0` and `libicen` respectively (table 1).

5. EVALUATION

5.1 Security Evaluation

Recall that we wish to guarantee both safety and liveness properties. We discuss both aspects separately.

5.1.1 *Safety Properties*

To ensure that our presented algorithm does guarantee state continuity, we developed a formal proof of correctness. While alternative formalizations (e.g., LS^2 [6]) may have reduced our workload, we formalized our system using rely-guarantee reasoning [12] as it enables explicit reasoning about attack steps. A machine-checked proof was created using the Coq proof assistant. The proof required 118 definitions, 201 lemmas and totals 37,726 lines. The extended version of this paper [27] discusses the proof in more detail.

5.1.2 *Liveness Properties*

In order to guarantee liveness, `libice0` and `libicen` must always be able to recover from a crash. We discuss how this is accomplished during their different phases.

`libice0`'s liveness properties.

An important distinction can be made based on the value of `tpm.mode`. This value indicates whether the algorithm has been initialized correctly. A crash before this value is set, will result in a re-execution of the initialization step. After setting this value, all crashes will result in the execution of the recovery step. To ensure that the initialization step may succeed eventually, we store the fresh cube on disk, take exclusive access of guarded memory, write the matching guards to guarded and TPM NVRAM memory and store cryptographic keys in the TPM chip *before* setting `tpm.mode`.

After initialization we may update the state or we have to recover the fresh state. In the former case we make sure to first store the cube before we update the content of guarded memory. Recovery of a state is more challenging as we have to modify the guards in both guarded and TPM NVRAM memory. After creating a new cube with the fresh state of the module and storing it on disk, we take exclusive access of guarded memory and write the new guard to it before we update TPM NVRAM memory. This has an important consequence: in case the system crashes during the execution of the recovery step before it is completed, the old guard may have been overwritten. This would prevent the re-execution of the recovery step. Therefore we require that this (public) guard is written to disk *before* the recovery step is called.

`libicen`'s liveness properties.

Ensuring liveness of `libicen`'s algorithm is straightforward as we only have to deal with two non-volatile data objects: cubes and the `ice0` module to store freshness information. For obvious reasons we ensure that new cubes are stored on disk first. `ice0`'s implementation and the `libice0` library it uses, guarantee that its state updates can be considered atomically and they are always retrievable.

5.2 Performance Evaluation

In this section we evaluate the performance of our prototype implementation. To compare the performance impact of a solid state drive (SSD) against a rotating hard drive (HDD), we used two machines with comparable hardware. The first machine, a Dell Latitude E6510, a mid-end consumer laptop, is equipped with an Intel Core i5 560M processor running at 2.67 GHz and 4 GiB of RAM. It is also equipped with a magnetic hard disk (HDD), a Broadcom TPMv1.2 chip and CMOS memory. The second testing laptop is a Dell Latitude E6520, has an Intel Core i5-2520M CPU running at 2.50GHz and is equipped with an SSD.

Hardware Benchmarks.

To better understand the performance cost of ICE compared to TPM operations, we performed 4 benchmarks on the Latitude E6510: read/write accessing TPM NVRAM, extending PCR registers and generating random numbers. To perform these tests, we developed small TPM appli-

[8] `https://sparrow.ece.cmu.edu/group/flicker.html`
[9] `http://polarssl.org/`

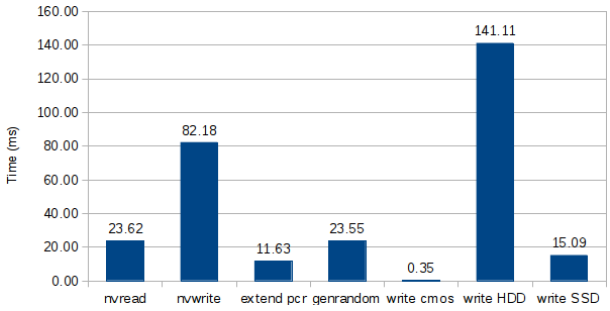

Figure 2: Microbenchmarks of various TPM operations show a significant difference in performance cost of CMOS and disk accesses. Where applicable, 128 bytes were transferred.

cations using the TrouSerS[10] open-source software stack. We also modified the tpm_tis driver to keep timing measurements. Each test was run 100 times and transferred 128 bytes to/from the TPM. Figure 2 displays the median time for each test. All operations take a significant amount of time to complete. Especially writing to TPM NVRAM takes 4x longer than reading from it. Related work shows similar results for TPM chips from other vendors [20].

We also performed a similar benchmark on CMOS memory. We performed 10,000 one-byte write operations and measured the time using the `rdtscp` instruction. Writing to CMOS takes about 3μs/byte, significantly faster than writing to TPM NVRAM. We attribute this difference to the fact that CMOS memory is connected to the SPI-bus [10] and does not require a heavy communication protocol as does the LPC-connected TPM chip.

Finally, we measured the median time of writing 10,000 128 bytes files to both HDD and SSD disks. As Figure 2 shows, accessing the SSD disk is 5.4 times faster than writing to TPM NVRAM. Writing to a magnetic disk is more costly.

Microbenchmarks.

To measure the performance of both `libice0` and `libicen` libraries, we implemented two modules. The first module implements a password verification function and limits the number of attempts that can be made before the user is locked out indefinitely. The benchmark provided this module with 10,000 wrong password guesses and measured the median time per guess. Measurements show (see Table 2) that for a single step only 0.06ms (0.43%) were spent on computation when the module was linked with the `libice0` library. When we used `libicen`'s services, two cubes need to be created and computation time increased to 0.13ms (0.71%). To securely write guards to CMOS memory, 0.33ms were spent (2.17% and 1.82% for `libice0` and `libicen` resp.). This shows a much higher cost to write guards to CMOS compared to calculation time. But most of the time was spent committing cubes to solid state disk (97.40% and 97.47% for `libice0` and `libicen` resp.). `libicen` does *not* spend twice the amount of time writing cubes to disk. Cubes only need to be committed before a guard is incremented. Hence, `libicen`'s cubes can be stored temporarily in memory and transferred to disk together with `ice0`'s new cube *without* modifying the algorithm (see listing 7, lines 3-4), reducing disk access times.

While most TPM chips NVRAM area is limited to 1,280 bytes [20], it could be used to provide (state-continuous) storage to a single module to avoid disk overhead. To show that such a module would still benefit from ICE, we implemented a second benchmark called Noop. It does not perform any computation but only stores a state of 1,280 bytes. As expected given the performance of SHA-512 and Intel's AES hardware support, the increase in computation cost is negligibly with only 0.01ms. As cubes are still smaller than disk sectors, costs of disk accesses are comparable to the Password benchmark. This totals the cost of storing new data in Noop at 15.05ms to 17.65ms for `libice0` and `libicen` resp; significantly faster than 82.18ms to access TPM NVRAM. Finally we performed these tests on the Latitude E6510 which is equipped with a magnetic HDD. As expected, the cost of writing cubes to disk increased significantly and now accounts for 99.63%-99.74%. For both benchmarks `libicen` consistently takes more time writing cubes to disk than `libice0`. We attribute this behavior to the way we implemented its write function: merging `ice0`'s and `libicen`'s cubes takes us 3 `write` system calls before system buffers are flushed.

Expected Impact of Dedicated Hardware.

These benchmarks show that only up to 0.14% of time is spent on computation. With dedicated hardware performance can be increased significantly.

Writing guards to CMOS memory is about 2.4 times more costly than computation and takes up to 0.31% of the time in case of a revolving HDD and up to 2.17% on our SSD testing platform. Hardware support for guarded memory, as described in detail in Section 3.1, would reduce overhead of this operation to almost zero.

But committing cubes to disk forms the real bottleneck, requiring up to 97.47% (for SSD) to 99.74% (for HDD) of the time. Recently Viking Technology [34] and Micron Technology [33] announced that they will ship capacitor-backed RAM to market. Operating similar to guarded memory, these hardware components contain fast, volatile memory that is written to flash memory when power is suddenly lost. Adding these hardware components to our system would eliminate disk access completely.

In summary, benchmarks show that our prototype implementation on commodity hardware already outperforms TPM NVRAM write operations by almost 5 times. Adding dedicated hardware support for guarded memory and capacitor-backed RAM, may even enable state updates 587 times faster than TPM NVRAM accesses!

6. RELATED WORK

With the exception of Parno et al. [20], state-continuous storage has largely been overlooked. Most research prototypes rely on a huge TCB or are vulnerable to crash attacks.

Hardware Modifications.

XOM [15] protects against an attacker that is able to snoop buses and modify memory by encrypting data and code before it is sent to memory. While it makes it significantly more difficult to successfully attack the system, Suh et al. [30] argue correctly that it is vulnerable to a memory replay attack where stale memory pages are returned to

[10] `http://trousers.sourceforge.net/`

	Password		Noop	
SSD (in ms)	`-lice0`	`-licen`	`-lice0`	`-licen`
computation	0.06	0.13	0.07	0.14
writing guard	0.33	0.33	0.33	0.33
writing cubes	14.61	17.42	14.65	17.19
total	15.00	17.87	15.05	17.65

HDD (in ms)	`-lice0`	`-licen`	`-lice0`	`-licen`
computation	0.06	0.12	0.07	0.13
writing guard	0.35	0.35	0.35	0.35
writing cubes	112.80	183.23	111.54	183.83
total	113.21	183.71	111.96	184.31

Table 2: Microbenchmarks for `libice0` and `libicen`

the processor. Their Aegis architecture mitigates this replay attack by storing hash trees of memory pages in a secure location. When a memory page is loaded into the processor's cache, its freshness is checked by recalculating and comparing the hash values. Subsequent research results also defend against replay attacks [4, 11, 37].

Memory replay attacks differ from rollback attacks in that memory contents is replayed *while* the system is up and running. This enables much easier security measures.

Schellekens et al. [24] propose an embedded-systems architecture to store a trusted module's persistent state in invasive-attack-resistent, non-volatile memory. Their solution implements a light-weight authenticated channel between the trusted module and non-volatile memory. Freshness of the stored data is guaranteed per read/write instruction and based on a monotonic counter. As their approach assumes that write instructions to non-volatile memory and increments of the monotonic counter are atomic, unexpected loss of power enables a rollback attack. We believe that their approach can be fixed by keeping a log of instructions in secure non-volatile memory that need to be completed in case power suddenly fails. On higher-end systems however, only the TPM NVRAM can be used for such purposes and their approach would lead to significant performance overhead. ICE, in contrast, is not affected by TPM performance.

Research Systems Isolating Persistent Storage.

Many architectures rely on a large TCB that includes isolation of persistent storage [7, 25, 31]. In such cases protection against rollback attacks are trivial: modules/programs can overwrite their state on disk. In practice however, software vulnerabilities in their TCB may be exploited and state-continuity support is hard to guarantee. These systems are also not able to defend against disk clone attacks. In contrast, ICE provides strong guarantees while only relying on a very limited TCB.

Protected-Module Architectures.

Many security architectures with minimal TCB have been proposed that only providing strong module isolation guarantees [3, 11, 16, 17, 23, 28]. Persistent storage can only be accessed via services provided by the untrusted operating system. None of them address the issue of state continuity.

Many of these systems can be adapted to use the state-continuity approach presented by Parno et al. [20]. This seminal work called Memoir, is to the best of our knowledge the first and only work that addresses the issue of state continuity in protected-module architectures. Based on Flicker [17], Memoir uses TPM NVRAM to store freshness information upon every state update. This significantly limits the applicability of their solution as NVRAM is slow and only required to support up to 100K writes. The authors acknowledge this constrained and propose two solutions: (1) adding capacitor-backed RAM to the TPM chip and (2) Memoir-Opt, an alternative approach that stores freshness information in (volatile) TPM PCR registers that are written to NVRAM when power is lost unexpectedly. Both solutions rely on an uninterruptible power source to safely store freshness when power suddenly fails. Failure in this mechanism can lead to a rollback attack. ICE, in contrast, is a passive state-continuity system that does not rely on an uninterruptible power source to guarantee security; detaching the capacitor would only prevent stateful modules from recovering their state but states could not be rolled back. Moreover, in ICE the speed of updates to state-continuous modules is only limited by the processor and (untrusted) non-volatile memory, not by the TPM chip.

Special-Purpose Applications.

Chun et al. proposed the creation of append-only memory [5] to prevent that nodes in a distributed system can make different statements to different nodes. An implementation with a minimal TCB was left as future work.

Levin et al. propose TrInc [14], a specialized system to attest successive monotonic counters, to achieve similar results. TrInc assumes a dedicated device that is able to locally store attestation requests of monotonic counters. After power was suddenly lost, clients can request the last signed attestations. This approach is similar to solutions were disk space is isolated, but incurs only a limited TCB. ICE provides a more generic, low-overhead alternative with only limited hardware modifications.

More recently Kotla et al. proposed a system [13] that allows offline data access while guaranteeing that (1) a user cannot deny offline accesses without failing an audit and (2) after proving that a user did not access the data, it cannot be accessed in the future. While their solution is interesting and does not require any software to be trusted, it only solves state-continuity in this specific setting.

7. CONCLUSION

Providing support for state continuity is challenging as including non-volatile memory on-chip requires modification of fabrication processes. But off-chip storage of freshness information can be slow (e.g. TPM NVRAM) or vulnerable to attack. We presented ICE, a state-continuous system and algorithm with two important properties: (1) only at boot time is the (slow) TPM chip accessed. State updates after the system booted only require updates to dedicated registers backed off-chip by a capacitor and non-volatile memory. (2) ICE is a passive security measure. An attacker interrupting the main power supply or any other source of power, cannot break state-continuity. We believe that the importance of ICE lies in the fact that it shows that with only limited and cheap hardware support, it enables the development of software-only implementations of trusted computing primitives. This presents an interesting direction for future versions or revisions of hardware security modules (e.g., the TPM) and may provide an interesting approach to increase security in low-end, resource-constrained applications.

Acknowledgments

The authors thank all reviewers and proofreaders of the paper for their useful comments. We also thank Frédéric Vogels and Dominique Devriese for their help with Coq.

This work has been supported in part by the Intel Lab's University Research Office. This research is also partially funded by the Research Fund KU Leuven, and by the EU FP7 project NESSoS. Raoul Strackx holds a PhD grant from the Agency for Innovation by Science and Technology in Flanders (IWT).

8. REFERENCES

[1] P. Agten, R. Strackx, B. Jacobs, and F. Piessens. Secure compilation to modern processors. In *CSF'12*.

[2] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for CPU based attestation and sealing. In *HASP'13*.

[3] A. Azab, P. Ning, and X. Zhang. SICE: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *CCS'11*.

[4] D. Champagne and R. Lee. Scalable architectural support for trusted software. In *HPCA'10*.

[5] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *OSR'07*.

[6] A. Datta, J. Franklin, D. Garg, and D. Kaynar. A logic of secure systems and its application to trusted computing. In *30th IEEE Symposium on Security and Privacy*, pages 221–236. IEEE, 2009.

[7] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *OSR'03*.

[8] J. Halderman, S. Schoen, N. Heninger, W. Clarkson, W. Paul, J. Calandrino, A. Feldman, J. Appelbaum, and E. Felten. Lest we remember: Cold boot attacks on encryption keys. In *USENIX'08*.

[9] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *HASP'13*.

[10] Intel Corporation. *Intel 6 Series Chipset and Intel C200 Series Chipset*, 2011.

[11] Intel Corporation. *Software Guard Extensions Programming Reference*, 2013.

[12] C. Jones. Tentative steps toward a development method for interfering programs.

[13] R. Kotla, T. Rodeheffer, I. Roy, P. Stuedi, and B. Wester. Pasture: secure offline data access using commodity trusted hardware. In *OSDI'12*.

[14] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. Trinc: Small trusted hardware for large distributed systems. In *NSDI'09*.

[15] D. Lie, T. Chandramohan, M. Mark, L. Patrick, B. Dan, M. John, and H. Mark. Architectural support for copy and tamper resistant software. In *ASPLOS'00*.

[16] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *S&P'10*.

[17] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *EuroSys'08*.

[18] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP'13*.

[19] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. V. Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *Usenix'13*.

[20] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical state continuity for protected modules. In *S&P'11*.

[21] M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens. Secure compilation to protected module architectures. In *Accepted for publication in ACM Transactions on Programming Languages and Systems (TOPLAS)*.

[22] M. Patrignani, D. Clarke, and F. Piessens. Secure Compilation of Object-Oriented Components to Protected Module Architectures. In *APLAS'13*.

[23] D. P. Sahita R, Warrier U. Protecting Critical Applications on Mobile Platforms. Intel.

[24] D. Schellekens, P. Tuyls, and B. Preneel. Embedded trusted computing with authenticated non-volatile memory. In *TRUST'08*.

[25] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. Reducing TCB complexity for security-sensitive applications: three case studies. In *EuroSys '06*.

[26] E. R. Sparks. A security assessment of trusted platform modules. Technical report.

[27] R. Strackx, B. Jacobs, and F. Piessens. ICE: A passive, high-speed, state-continuity scheme (extended version). Technical report, KU Leuven, Sept. 2014.

[28] R. Strackx and F. Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *CCS'12*.

[29] R. Strackx, F. Piessens, and B. Preneel. Efficient Isolation of Trusted Subsystems in Embedded Systems. In *SecureComm'10*.

[30] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *ICS'03*.

[31] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *OSDI'06*.

[32] C. Tarnovsky. Deconstructing a "secure" processor. In *Black Hat'10*.

[33] M. Technology. Hybrid memory - bridging the gap between DRAM speed and NAND nonvolatility.

[34] V. Technology. NV-DIMM: Achieving greater ROI from SSDs. Technical report.

[35] Trusted Computing Group. Design Principles Specification Version 1.2. 2011.

[36] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta. Design, implementation and verification of an extensible and modular hypervisor framework. In *S&P'13*.

[37] P. Williams and R. Boivie. CPU support for secure executables. In *TRUST'11*.

[38] J. Winter and K. Dietrich. A hijacker's guide to the LPC bus. In *EuroPKI'11*.